M.Sc. in High-Performance Computing
5614. C++ Programming
Assignment 5

R. Morrin <rmorrin@maths.tcd.ie>

March 3, 2025

Instructions

- Gather all your code and pdf/text files into a single tar-ball.
- Submit this tar-ball via `Blackboard` before 23:59 Fri 11th Apr.
- Any questions, please ask me in class or email me at rmorrin@maths.tcd.ie. Don't wait until right before the deadline!
- Late submissions without prior arrangement or a valid explanation will result in reduced marks. Just let me know if you are having difficulty.
- All non-code answers should be submitted in a single pdf or text file. If you write and scan in, please make sure I can read it!
- The non-code questions only need short answers - not an essay!
- Read notes at end of this document.
- Remember to use comments when appropriate.

Assignment 5. Smart pointers and Monte Carlo

Q1) Makefile (5%)

Write a `Makefile` to take care of the rest of the questions. You should just need to run `make` to compile everything up.

Q2) Unique Pointers (30%)

`unique_ptr`

For the first question you will write your own smart pointer class `HPC::unique_ptr`. This will be a templated class which will be used to model a pointer which has sole ownership and responsibility for an underlying managed object.

You will write a class for `HPC::unique_ptr`. The class will contain a `private` member variable `T *managed_object`. You can download the skeleton for the class definition in unique_ptr.h. The skeleton contains some declarations and also the destructor definition. There are other member function which are not declared.

Write all member function definitions outside the class definition.

The class should:

- not be *DefaultConstructible*[1].
- not be *CopyConstructible*[2].
- not be *CopyAssignable*[3].
- be *MoveConstructible*[4].
- be *MoveAssignable*[5].
- be *Destructible*[6].

The above are just a technical way of saying that it won't have a default constructor (`std::unique_ptr` does), it will have a destructor that releases all resources, it will implement the move semantics, but not implement the copy semantics.

Download the `main` function in assignment5a.cc. This should run as is. Some lines are commented. If you uncomment them, you should see similar errors to the comments above the lines.

The constructor for the class takes a raw pointer and initialises the value of `managed_object` to this value.

```
unique_ptr (T*);
```

(a) Implement the Rule of Five Defaults for this `class`

As you've covered in earlier assignments and lectures, the Rule of Five Defaults[7] says to declare (or `delete`) or `default` the destructor, move semantics, and copy semantics, for every class you write.[8]

[1] `https://en.cppreference.com/w/cpp/named_req/DefaultConstructible`
[2] `https://en.cppreference.com/w/cpp/named_req/CopyConstructible`
[3] https://en.cppreference.com/w/cpp/named_req/CopyAssignable
[4] `https://en.cppreference.com/w/cpp/named_req/MoveConstructible`
[5] `https://en.cppreference.com/w/cpp/named_req/MoveAssignable`
[6] `https://en.cppreference.com/w/cpp/named_req/Destructible`
[7] `http://scottmeyers.blogspot.com/2014/03/a-concern-about-rule-of-zero.html`
[8] This is a variation of `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c21-if-you-define-or-delete-any-copy-move-or-destructor-function-define-or-delete-them-all`

Implement this rule for your `HPC::unique_ptr` class. For the member function definitions you write, print a simple message to show where they are called. See the example output to give you an idea. Again, you wouldn't do this in real-world code.

For move assignment, think about what you need to do given that the target `HPC::unique_ptr` might have already been managing a different object. You won't be able to access (or delete) once you have overwritten that original object, so you need to clean it up now. Also consider whether it might be possible for a user (accidentally or otherwise) to do something like

```
HPC::unique_ptr D {new double {1.0}}
D = std::move(D);
```

(b) Other member functions

First, overload the dereference operator to return a reference to the underlying managed object.

```
T& operator*();
```

(This definition body only needs one line . . . It doesn't need to handle the case where the underlying raw pointer is `nullptr` . . . FYI, `std::unique_ptr` doesn't handle this case either).

You will also need a corresponding const member function so that you can access a const object of the class.

```
T operator*() const;
```

Also implement the definition for the member function:

- `T* get();`. This returns the raw pointer to the `managed_object`.
- `T* release();`. This will release the ownership of the `managed_object` (i.e reset the internal raw pointer to `nullptr`) and return the raw pointer to the previously managed object. It does not delete the managed object.

Output

```
$ ./assignment5a
Constructing HPC::unique_ptr for type Pd with value 1
Constructing HPC::unique_ptr for type Pd with value 2
Constructing HPC::unique_ptr for type Pd with value 3
Move Constructing HPC::unique_ptr for type Pd with value 3
E = 3
E = 4
D = 4
Deleting HPC::unique_ptr for type Pd containing null pointer
Deleting HPC::unique_ptr for type Pd with value 4
Deleting HPC::unique_ptr for type Pd with value 2
Deleting HPC::unique_ptr for type Pd with value 1
```

Q3) Shared Pointer (30%)

For this question, you will implement your own shared pointer to model shared ownership of an object. Your code should run with assignment5b.cc. Download the skeleton of the class in shared_ptr.h. Write the definitions for the member functions outside the class definition.

Note that this class contains the following two private member variables

```
T* managed_object;
unsigned int* counter;
```

where `counter` keeps a count on how many `HPC::shared_ptr`s point to the underlying `managed_object`.

When you copy a `HPC::shared_ptr`, the copy should point to the same `managed_object` and `counter` should be increased.

Constructor and Destructor

Like the `HPC::unique_ptr`, the constructor takes a raw pointer, sets the value of the `managed_object` and then allocates and sets the value of `counter` to 1. For the destructor, if `counter` > 1 then it should simply be decremented. if it $= 1$ then the memory should be deleted.

```
shared_ptr (T*);
~shared_ptr ();
```

Move and Copy Semantics

The `HPC::shared_ptr` should be copyable and moveable. Implement the copy and move semantics.

For the assignment operators, think about how to handle the case where the target `HPC::shared_ptr` is already pointing to a different `managed_object`. This is not as straightforward as for the move assignment in the case of `HPC::unique_ptr` above.

Other functions

Write the function definitions for:

```
T* get();
```

This should return the address of the managed object.

```
T* release();
```

This should release ownership from the current object and returns the pointer to the managed object. Ownership is released by setting `managed_object` equal to `nullptr`. It should not delete the managed object.

```
unsigned int use_count();
```

This should return the number of `HPC::shared_ptr` currently sharing ownership of the underlying `managed_object`.

Output

Output

```
$ ./assignment5b
```

```
Constructing A
Constructing HPC::shared_ptr for type Pd with value 10
A.get() = 0x5a20f2cca6c0
A.use_count() = 1
*A = 10
```

```
Copy Constructing B from A
Copy Constructing HPC::shared_ptr for type Pd with value 10
New Reference Counter =  2
B.get() = 0x5a20f2cca6c0
B.use_count() = 2
*B = 10
*A = 100
```

```
Constructing C
Constructing HPC::shared_ptr for type Pd with value 20
Before Assignment:
C.get() = 0x5a20f2cca700
C.use_count() = 1
*C = 20
Copy Assignment C from A
Copy assignment operator
After Assignment:
C.get() = 0x5a20f2cca6c0
C.use_count() = 3
*C = 100
```

```
End of main()
Deleting HPC::shared_ptr for type Pd with value 100
New Reference Counter = 2
Deleting HPC::shared_ptr for type Pd with value 100
New Reference Counter = 1
Deleting HPC::shared_ptr for type Pd with value 100
Reference Counter = 0 .... Deleting managed object
```

Q4) Lambda expressions and Functors (10%)

(i) Write an equivalent lambda expression.

Download assignment5c.cc. This code contains a struct with an overloaded function call operator. This functor is used in the `std::transform` algorithm to modify values of a vector in place.

Write a lambda expression that you can pass into `std::transform` that will achieve the same thing.

assignment5c.cc

```
struct transform_op {
    double operator()(double x){
        return (x * std::log(x));
    }
};

int main(void)
{
    std::vector<double> A(10000);
    std::iota(A.begin(), A.end(), 1.0);

    std::transform(std::execution::par, A.begin(), A.end()\
                        , A.begin(), transform_op {});

    for (auto const& a : A) {
        std::cout << a << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

Note that you will probably have to link to a threading library to get this to compile, e.g you will need `-ltbb` on `chuck`. This is because the code uses the execution policy overload version for the algorithm.

(ii) Write an equivalent functor.

Download assignment5d.cc. This time you will write a functor `is_lower_than` which will do the equivalent of the given lambda expression.

This code uses `std::remove_if` which removes any element in the container which satisfies the condition. That function does not truncate the container - it simply replaces the elements at the start of the container with the surviving elements. It returns an iterator to (one after) the end of the surviving list. So we then call the `erase` member function to remove the unneeded space.

assignment5d.cc

```
/// TODO:
/// Write functor is_lower_than
///

int main(void)
{
    std::vector V {-1.2, 5.4, 12.2, -3.9, -0.4, 13.1, 0.5};

    const double cutoff_value {1.1};

    auto lambda = [cutoff_value](double testval){
        return testval <  cutoff_value;
    };

    auto new_it = std::remove_if(std::begin(V), std::end(V), lambda);
    V.erase(new_it, std::end(V));

    // print filtered vector to screen
    for(auto const& v : V){
        std::cout << v << " ";
    }
    std::cout << '\n';


    return 0;
}
```

Q4) Random Numbers (10%)

This part will use random numbers and `std::valarrays`. This is quite short. You can write it all in a `main` function in assignment5e.cc that you will write

from scratch. You only need about 12-15 lines of code in your `main`. Don't use any loops.

What you will do is estimate the mean and variance from a lognormal distribution by pulling numbers from it and then calculating the quantities from those numbers. You don't need to worry about seeding the random number engine. Use `ranlux48` (but it doesn't change the code regardless of what engine is chosen). You will have one `std::valarray<double>` to hold the random variates. You just need one.

Create a `std::lognormal_distribution` with parameters `{0,1}`. Use `std::bind` to bind this to the engine you declared. See slide 15 of L14 if you are stuck.

Then use `std::generate` to populate your `std::valarray`. See slide 18 of L14. `std::valarray` class does not have its own iterators, but `std::begin` and `std::end` are specialised to provide them for you.

Then calculate the first moment of the distribution using the inbuilt `std::valarray` member functions. One line.

Then square each of the values in the `std::valarray` (Take advantage of the `operator*` overload for `std::valarray`). Again, one line/

Then calculate the second moment of the distribution. Again one line.

Then calculate the variance from the two moments.

Estimate from running for 10000000 random variates

```
$ ./assignment5e
Avg = 1.64941
Variance = 4.68379
```

Q5) Written Questions. (15%)

Smart Pointers

(i)

Why might you prefer to use smart pointers over raw pointers? Why might you prefer to use a `std::unique_ptr` instead of a `shared_ptr` even if the latter would also work correctly in your code?

(ii)

The following code compiles without warning, but results in undefined behaviour. Explain the issue.

```
int main()
{
    auto p {new double};
    std::shared_ptr<double> A {p};
    std::shared_ptr<double> B {p};

    return 0;
}
```

(iii)

Explain in your own words what a `std::weak_ptr` is. (A sentence or two should be enough)

Explain how you could use `std::shared_ptr` and `std::weak_ptr` to help avoid dangling pointers.

Notes:

- For your own code, prefer to use smart pointers over raw pointers.
- Put some sanity checks into your code. Regular checks are fine - you don't need to worry about exceptions or asserts etc. Just some simple logic checks such as bounds checking etc. However, if you encounter an error for this assignment, throw an error message rather than an exit.
- Use member list initialisation where possible.
- Use universal initialisation syntax {} where you can rather than = or () notations.
- 10% of the respective marks available for each of the coding questions will be given for comments. You do not need to go overboard and please make sure the comments are relevant. i.e. you don't need to say obvious things like `int A; /*This is an integer */`. Note that in the lecture slides, I sometimes include comments such as `/* Function Prototype */` - but this is just for the purposes of the lectures. This would not necessarily be a helpful comment on its own in a proper program. Strive to make your code as self-documenting as possible by using appropriate names for variables and functions.
- To create your tarball, `cp` everything you want to submit into a directory e.g. `Assignment5`, and then change directory so that you are in the parent directory of `Assignment5`.

  `tar -cvf [username].assignment5.tar Assignment5`

  where you replace `[username]` with your own username.